Institute for Informatics of the Ludwig-Maximilians-Universität München
Prof. Dr. Jasmin Blanchette
Xavier Généreux
Balazs Toth

Summer semester 2024
25.07.2024

Final Examination in the Course

Interactive Theorem Proving

You have **120 minutes** at your disposal. Written or electronic aids are not permitted. Carrying electronic devices, even switched off, will be considered cheating.

Write your full name and matriculation number clearly legible on this cover sheet, as well as your name in the header on each sheet. Hand in all sheets. Leave them stapled together. Use only **pens** and **neither** the color **red nor green**.

Check that you have received all the sheets. Guidelines for writing pen-and-paper proofs are given on **page 1**. Questions can be found on **pages 2–16**. You may use the back of the sheets for secondary calculations. If you use the back of a sheet to answer, clearly mark what belongs to which question and indicate in the corresponding question where all parts of your answer can be found. Cross out everything that should not be graded.

There are 6 questions for a total of 100 points. The subquestions can be completed independently of each other.

With your signature, you confirm that you are in sufficiently good health at the beginning of the examination and that you accept this examination bindingly.

Last name:

First name:

Matriculation number:

Program of study:

□ Please check with an X *only* if the exam should be voided and not graded.

Bitte *nur* ankreuzen, wenn die Klausur entwertet und nicht korrigiert werden soll.

Hierby I confirm the correctness of the above information: _____

Signature

Please leave the following table blank:

Question	1	2	3	4	5	6	$\sum$
Points	20	25	25	8	12	10	100
Score							

Guidelines for Paper Proofs

We expect detailed, rigorous, mathematical proofs, but we do not ask you to write Lean proofs. You are welcome to use standard mathematical notation or Lean structured commands (e.g., `assume`, `have`, `show`, `calc`). You can also use tactical proofs (e.g., `intro`, `apply`), but then please indicate some of the intermediate goals, so that we can follow the chain of reasoning.

Major proof steps, including applications of induction and invocation of the induction hypothesis, must be stated explicitly. For each case of a proof by induction, you must list the induction hypotheses assumed (if any) and the goal to be proved. Minor proof steps corresponding to `refl`, `simp`, or `linarith` need not be justified if you think they are obvious, but you should mention which key lemmas they depend on. You should be explicit whenever you use a function definition or an introduction rule for an inductive predicate.

Question 1 (Types and Terms): **(20 points)**

a) Recall the following simplified typing rules for Lean's dependent type theory:

$$\frac{}{\texttt{C} \vdash \texttt{c} : \sigma}\ \textsc{Cst} \quad \text{if } \texttt{c} \text{ is globally declared with type } \sigma$$

$$\frac{}{\texttt{C} \vdash \texttt{x} : \sigma}\ \textsc{Var} \quad \text{if } \texttt{x} : \sigma \text{ is the rightmost occurrence of } \texttt{x} \text{ in } \texttt{C}$$

$$\frac{\texttt{C} \vdash \texttt{t} : (\texttt{x} : \sigma) \to \tau\texttt{[x]} \quad \texttt{C} \vdash \texttt{u} : \sigma}{\texttt{C} \vdash \texttt{t u} : \tau\texttt{[u]}}\ \textsc{App}'$$

$$\frac{\texttt{C, x} : \sigma \vdash \texttt{t} : \tau\texttt{[x]}}{\texttt{C} \vdash \texttt{(fun x} : \sigma \mapsto \texttt{t)} : \texttt{(x} : \sigma\texttt{)} \to \tau\texttt{[x]}}\ \textsc{Fun}'$$

Let $\texttt{a} : \mathbb{N}$, $\texttt{f} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$, $\texttt{g} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, and $\texttt{h} : (\texttt{y} : \mathbb{N}) \to \{\texttt{x} : \mathbb{N} \texttt{ // x < y}\}$ be globally declared constants. What is the type of the following two Lean terms? Give in each case a typing derivation as justification for the type.

(i) `h a`

(ii) `f (fun x` $: \mathbb{N} \mapsto$ `g x x)`

b) Let α, β, and γ be Lean types. Give an inhabitant for each of the following types:

- $\alpha \to \alpha$

- $(\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta$

- $(\gamma \to (\alpha \to \beta) \to \alpha) \to \gamma \to \beta \to \alpha$

Question 2 (Functional Programming): **(25 points)**

a) Consider the following Lean function definition:

```
def surround {α : Type} (a : α) : List α → List α
  | []      => [a]
  | b :: bs => a :: b :: surround a bs
```

(i) Compute the value of `surround 0 [1, 2, 3]`.

(ii) Prove the following Lean theorem. Make sure to follow the proof guidelines given on page 1.

```
theorem map_surround {α β : Type} (f : α → β) (x : α) (ys : List α) :
  List.map f (surround x ys) = surround (f x) (List.map f ys)
```

(iii) Prove the following Lean theorem. Make sure to follow the proof guidelines given on page 1.

```
theorem length_surround {α : Type} (x : α) (ys : List α) :
  List.length (surround x ys) = 2 * List.length ys + 1
```

b) Define a Lean function `suffixes` that takes a list and that returns a list of all suffixes of that list, including the list itself, from longest to shortest. For example, `suffixes [1, 2, 3]` should evaluate to `[[1, 2, 3], [2, 3], [3], []]`.

Question 3 (Inductive Predicates): (25 points)

a) Consider the following Lean inductive predicate, which determines whether a list has odd length:

```
inductive OddLength {α : Type} : List α → Prop
  | singleton (x : α) :
    OddLength [x]
  | add_two (x y : α) {xs : List α} :
    OddLength xs → OddLength (x :: y :: xs)
```

For example, `OddLength [1, 2, 3]` holds, whereas `OddLength [4, 5]` does not.

Prove the following Lean theorem about `OddLength`. Make sure to follow the proof guidelines given on page 1.

```
theorem OddLength_map {α β : Type} (f : α → β) (xs : List α)
    (hxs : OddLength xs) :
  OddLength (List.map f xs)
```

b) Define an inductive predicate `InList` in Lean that takes a value `x` of the polymorphic type α and a list `ys` over α as arguments and that holds if `x` occurs in `ys`.

c) Consider the following Lean type of binary trees and a helper function that counts the number of nodes in a given tree:

```
inductive Tree (α : Type) : Type
  | nil  : Tree α
  | node : α → Tree α → Tree α → Tree α

def numNodes {α : Type} : Tree α → ℕ
  | Tree.nil        => 0
  | Tree.node a l r => numNodes l + numNodes r + 1
```

Define an inductive predicate `RightHeavy` in Lean that takes a tree as argument and that holds if the tree enjoys *either* of the following properties:

- The tree is of the form `Tree.nil`.
- The tree is of the form `Tree.node a l r`, the number of nodes in `l` is less than or equal to the number of nodes in `r`, and both `l` and `r` recursively satisfy `RightHeavy`.

Question 4 (Metaprogramming): **(8 points)**

Consider the following custom Lean tactic:

```
macro "mystery" : tactic =>
  `(tactic| repeat' first
              | assumption
              | intro _
              | apply And.intro
              | apply Iff.intro)
```

a) Briefly explain what the `mystery` tactic does. You may assume that we already know what `assumption`, `intro`, and `apply` does.

b) In the following Lean code fragment, the `mystery` tactic is applied to massage the goal:

```
theorem abac (a b c : Prop) :
  a → b ∧ a ∧ ¬ c :=
  by
    mystery
```

The proof state before invoking `mystery` is

```
a b c : Prop
⊢ a → b ∧ a ∧ ¬c
```

Give the proof state *after* invoking `mystery`. Make sure to include all subgoals.

Question 5 (Operational Semantics): (12 points)

The IF0 programming language is similar to WHILE, with two differences. First, the `while`–`do` statement is omitted. Second, the `if`–`then`–`else` statement is replaced by `if_zero`–`then`–`else`. For example, the program

```
if_zero m * n then
  x := 0
else
  y := 1
```

executes `x := 0` if the condition `m * n = 0` holds when entering the construct; otherwise, it executes `y := 1`.

In Lean, IF0 would be modeled by the following datatype:

```
inductive Stmt : Type where
  | skip   : Stmt
  | assign : String → (State → ℕ) → Stmt
  | seq    : Stmt → Stmt → Stmt
  | ifZero : (State → ℕ) → Stmt → Stmt → Stmt

infixr:90 "; " => Stmt.seq
```

a) Complete the following specification of a big-step semantics for IF0 in Lean by giving the missing derivation rules for `seq` and `if_zero`.

$$\frac{}{\texttt{(skip, s)} \Longrightarrow \texttt{s}}\ \textsc{Skip}$$

$$\frac{}{\texttt{(x := a, s)} \Longrightarrow \texttt{s[x} \mapsto \texttt{s(a)]}}\ \textsc{Assign}$$

b) Complete the following Lean definition of an inductive predicate that encodes the big-step semantics you specified in subquestion a) above.

```
inductive BigStep : Stmt × State → State → Prop where
  | skip (s) :
    BigStep (Stmt.skip, s) s
  | assign (x a s) :
    BigStep (Stmt.assign x a, s) (s[x ↦ a s])
```

Question 6 (Mathematics): **(10 points)**

a) Let σ `: Type 2` and τ `: Type 3` be Lean types. Give the type of each of the following Lean terms.

`fun x :` $\sigma \mapsto$ `x`

`Type 5`

$\sigma \to \tau$

`fun` α `: Type` $\mapsto$ `Option (`$\alpha \times \alpha$`)`

b) A *bag* is a collection of elements that allows multiple (but finitely many) occurrences of its elements. For example, the bag `{2, 7}` is equal to the bag `{7, 2}` but different from `{2, 7, 7}`.

Finite bags can be defined as a quotient over lists. We start with the type `List` α of finite lists and consider only the number of occurrences of elements in the lists, ignoring the order in which elements occur. Following this scheme, `[2, 7, 7]`, `[7, 2, 7]`, and `[7, 7, 2]` would be three equally valid representations of the bag `{2, 7, 7}`.

The `List.count` function returns the number of occurrences of an element in a list. Since it uses equality on elements of type α, it requires α to belong to the `BEq` (Boolean equality) type class. For this reason, the definitions below take `[BEq` α`]` as a type class argument.

```
instance Bag.Setoid (α : Type) [BEq α] : Setoid (List α) :=
{ r     := fun as bs ↦ ∀x, List.count x as = List.count x bs
  iseqv :=
    { refl  := by simp
      symm  := by aesop
      trans := by aesop } }

def Bag (α : Type) [BEq α] : Type :=
  Quotient (Bag.Setoid α)

def Bag.mk {α : Type} [BEq α] : List α → Bag α :=
  Quotient.mk (Bag.Setoid α)
```

Complete the following two definitions.

```
def Bag.empty {α : Type} [BEq α] : Bag α :=

def Bag.singleton {α : Type} [BEq α] (a : α) : Bag α :=
```

c) Consider the following Lean type classes:

```
universe u

class Mul (α : Type u) : Type u where
  mul : α → α → α

infixl:70 " * " => Mul.mul

class Semigroup (α : Type u)
  extends Mul α : Type u where
  mul_assoc : ∀a b c : α, a * b * c = a * (b * c)
```

The `Semigroup` type class provides an associative binary operator called `mul` abbreviated as `*`.

The concatenation operator `++` on lists is an associative binary operator. Exploit this fact to fill in the gap in the following type class instantiation:

```
instance List.Semigroup {α : Type} : Semigroup (List α) :=
  { mul       :=
    mul_assoc := by simp }
```